

Appendix A: Source Code

The ANSI C source code for the functions *f*, *g*, *d* and *h* is given below, in which each function is implemented using a macro. In these definitions, *a* and *b* are 32-bit unsigned integers (or `uint32_ts`, in POSIX terminology), and the function *F* is represented by the array `F[256]` of 32-bit unsigned integers. The macros `ROT8` and `ROT24` implement rotation by eight bits and twenty-four bits, respectively, where the direction of rotation is towards the most significant bit.

```
10  #define ROT8(x)  (((x) << 8) | ((x) >> 24))
#define ROT24(x) (((x) << 24) | ((x) >> 8))

#define f(x, y, z, F) ( \
    z += z, \
    y = ROT24(y), \
    x ^= F[x & 0xFF], \
    y ^= F[y & 0xFF], \
    y = ROT24(y), \
    x = ROT8(x), \
    x ^= F[x & 0xFF], \
    y ^= F[y & 0xFF], \
    x = ROT8(x) \
)

25  #define g(x, y, z, F) ( \
    z += (z+1), \
    x = ~x; \
    x = ROT24(x), \
    x ^= F[x & 0xFF], \
    y ^= F[y & 0xFF], \
    x = ROT24(x), \
    y = ROT8(y), \
    x ^= F[x & 0xFF], \
    y ^= F[y & 0xFF], \
    y = ROT8(y) \
)

35  #define d(x, y, z) ( \
    x += z, \
    y += x, \
    x += y \
)

#define h(a, b) (a ^ b)
```

The source code to produce the j th word of output (that is, bytes $4j$ through $4j+3$) is given below, where j is represented by the variable `leaf_num`.

```
5  uint32_t
  leviathan_output(int leaf_num) {
    int i;
    uint32_t x, y, z;

10   i = 1 << (LEVIATHAN_HEIGHT-1);
    x = y = 0;
    z = 1;

15   while (i > 0) {
      d(x, y, z);
      if (i & leaf) {
        g(x, y, z, F); /* right */
      } else {
        f(x, y, z, F); /* left */
      }
      i >>= 1;
    }

25   return h(x, y);
  }
```

Source code for an embodiment of a key setup routine follows. Here, `key` is a pointer to an unsigned character string of length `bytes_in_key`, and `F` is an array of `TABLE_SIZE` words.

```
#define TABLE_SIZE 256
#define NUM_PASSES 2

35  void init_leviathan_key(const unsigned char *key,
                           size_t bytes_in_key, word *F) {
    int i, j, k, index;
    word tmp;

40  for (i=0; i<TABLE_SIZE; i++)
    F[i] = 0;

    /*
45   * Each iteration of this loop we form the permutation of one line
   * (and, incidentally, also permute previously formed lines)
```

```

        */

5      for (j=0; j<4; j++) {
        /*
         * Initialize the new line to the identity permutation, and
         * shift the existing lines over one
         */
10     for (i=0; i<TABLE_SIZE; i++)
         F[i] = F[i] * TABLE_SIZE + i;

15     /*
         * Initialize index to a line-dependant value, so that the
         * four lines will get distinct permutations
         */
20     index = j;

        /*
         * Do the byte-swapping NUM_PASSES times, using the new
         * line as the index
         */
25     for (k=0; k<NUM_PASSES; k++) {
        for (i=0; i < TABLE_SIZE; i++) {
            index += (key[i % bytes_in_key] + F[i]);
            index &= (TABLE_SIZE-1);
            tmp = F[i];
            F[i] = F[index];
            F[index] = tmp;
        }
    }
30 }

        /*
         * Finally, set S0 equal to the xor of itself with the
         * identity permutation, so that (S0[x] ^ x) is a permutation.
         */
35     for (i=0; i < TABLE_SIZE; i++)
         F[i] ^= i;

40 }

```